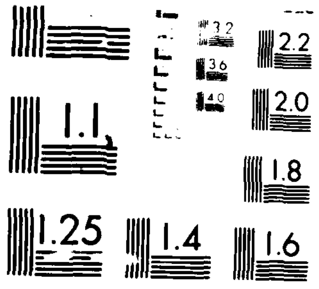
AD-A178 837 FORMAL MODELS OF HARDWARE AND THEIR APPLICATIONS TO VLSI DESIGN AUTOMATIO..(U) UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF ELECTRI.. A C PARKER 1/1

UNCLASSIFIED 24 DEC 86 ARO-20637.17-EL DAAG29-83-K-0147 F/G 9/5 NL

END
DATE FILMED
5-87



3.2
3.6
4.0
2.2
2.0
1.1
1.8
1.25
1.4
1.6

ARO 20637.17-EL

Formal Models of Hardware and Their Application to VLSI Design Automation

Final Report

Alice C. Parker

December, 1986

U.S. Army Research Office

Contract No. DAAG29-83-k-0147

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

87

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A178837

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 20637.17-EL
6a. NAME OF PERFORMING ORGANIZATION University of Southern California	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION Office of Naval Research	
6c. ADDRESS (City, State and ZIP Code) Department of Engineering Engineering-Systems, SAL-300 Los Angeles, California 90089-0781	7b. ADDRESS (City, State and ZIP Code) 1030 East Green Street Pasadena, California 91106-2485
8a. NAME OF FUNDING/SPONSORING ORGANIZATION US Army Research Office	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAG29-83-K-0147	
8c. ADDRESS (City, State and ZIP Code) US Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

11. TITLE (Include Security Classification)
FORMAL MODELS OF HARDWARE AND THEIR APPLICATION TO VLSI DESIGN AUTOMATION (Unclassified)

12. PERSONAL AUTHOR(S)
Alice C. Parker

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 09/83 TO 09/86	86/12/24	79

16. SUPPLEMENTARY NOTATION
The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Synthesis, pipelining, clocking hardware, optimization, area estimation, wiring, interconnect, behavior, data structure, design automation, computer aided design.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This final report describes research in high-level synthesis, and an associated problem, area estimation of integrated circuits. The approach taken is to create formal models of the problem being solved. Four major research results have been produced. First, an accurate technique for estimation of integrated circuit layout area from cell information has been developed. Second, optimal clocking scheme synthesis has been automated. Third, programs to design pipelined and non-pipelined data paths have been developed. Fourth, register allocation of the data paths has also been automated. In addition, a representation for design information which was produced under a previous contract has been used for a number of applications.

This research forms part of the ADAM Advanced Design AutoMation System under construction at the University of Southern California.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

Formal Models of Hardware and Their Application to VLSI Design Automation

Final Report

Alice C. Parker

December, 1986

U.S. Army Research Office

Contract No. DAAG29-83-k-0147

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

Table of Contents

A-1

1. ABSTRACT

This final report describes research in high-level synthesis, and an associated problem, area estimation of integrated circuits. The approach taken is to create formal models of the problems being solved. Four major research results have been produced. First, an accurate technique for estimation of integrated circuit layout area from cell information has been developed. Second, optimal clocking scheme synthesis has been automated. Third, programs to design pipelined and non-pipelined data paths have been developed. Fourth, register allocation of the data paths has also been automated. In addition, a representation for design information which was produced under a previous contract has been used for a number of applications.

This research forms part of the ADAM Advanced Design AutoMation system under construction at the University of Southern California.

2. INTRODUCTION

The focus of the research described in this final report has been on synthesizing hardware automatically from specifications of the required behavior. In order to perform this synthesis task properly, estimates of the silicon chip area required must be available. In addition, design data must be represented in a manner that can be manipulated easily by the synthesis programs. The specific problems studied under this contract include

1. techniques to perform area estimation from high level specifications,

2. methods to generate hardware automatically from behavioral specifications, while meeting timing and cost constraints, and,

3. applying models for representing hardware to support synthesis and verification.

The approaches to each of these problems, and results obtained, will now be described. A list of publications and supported personnel follows the research summary. Finally, program documentation for the synthesis and area estimation programs is contained in the appendices.

This research forms part of the ADAM Advanced Design AutoMation system under construction at the University of Southern California.

3. SUMMARY OF RESEARCH RESULTS

This section summarizes research in area estimation, synthesis, and application of hardware representations.

3.1. Area Estimation

The area estimation research has three components:

- statistical estimations of the channel capacities of gate array layouts,
- estimating the area of standard cell layouts, and
- estimating the area and performance of pipelined data paths from the behavior.

Gate-Array Area Estimation

The gate array wiring space estimation results allow us to estimate individual channel capacities on a gate array chip. These estimates can then be used to choose actual channel widths, and the probability of successfully routing these chips (routability) can also be estimated. Also in [19] we provide mathematical models to obtain quantitative measures for assessing the quality of the placement and routing solutions and estimating these measures *a priori*.

A gate array chip is modeled as a two dimensional lattice of points with the number of wires emerging from a point being a random variable following the Poisson distribution. Wires at the intersection of a horizontal and vertical channel are classified as belonging to one of six different types. The dimensions of the routing channel are defined as functions of these random variables.

We present probabilistic models of routing on master slice ICs for estimating three important measures of placement, namely, average wire lengths, wiring area, and routability. In specific, we present simple and computationally efficient methods for obtaining estimates of the dimensions of the individual routing channels. Additionally, asymptotic properties of these estimates are discused.

Next, the relationship between wire length distributions and partitioning is addressed. In particular we show that the empirical rule known as Rent's rule that characterizes "well" partitioned layouts corresponds to a family of wire length distributions known as the Weibul family. In fact any such relation, i.e. between the number of pins available on a module and the number of circuits that can be placed on the module, completely determines the distribution of wire lengths.

Finally, the question of routability is addressed. That is given a placement and the amount of wiring space, we ask what percentage of connections can be successfully completed at a given level of confidence. Since the dimensions of the routing channels are random variables, routability is defined in terms of the distribution of these random variables. Exact formulas for computing the routability of each channel are presented. Additionally, asymptotic formulas (ie. as the chip size becomes large) for routability are derived. Finally, computational results using the exact and asymtotic formulas for chips of various sizes are presented. Some of these results have been published in [20], [18].

Standard-Cell Area Estimation

Work has been completed on area estimation of standard cell IC's [6], [5]. The main focus is on establishing measures of net congestion in the intervening routing channels between rolls of standard cells. A simple empirical model was developed for that purpose. The model assumes the existence of relations between rows of standard cells as partitions and the number of nets which connect them to other rows or pads. The relations are similar in concept to Rent's rule. Experiments were done with the aim of investigation the existence of such relations in actual layouts. Some empirical evidence that such relations exist was found. Another problem researched is statistical modelling of row sizes with the aim of estimating the size of the widest row, which, in turn determines the width of a standard cell block. The method of row folding for placement of standard cells is used to model the variation in total routing track demand as the number of rows is increased and the aspect ratio is changed.

PLEST, a program for estimating the area of standard cell layouts has been written as part of the more general ARREST area estimator. PLEST is based on a probabilistic model for placement of logic. Given various design parameters, PLEST generates a range of estimates for the possible shapes of the block layout. The program was applied to a set of six layouts. The estimated chip area is, for all six chips, within 10% of the measured area. Documentation for PLEST is attached and PLEST is available on tape.

Average wire length estimation is an important parameter in our estimation model. We investigated the validity of Rent's rule for standard cell designs. We developed a scheme for estimating the Rent parameters and for using Rent's rule to estimate the average wire length. Comparison with real chip layout data will be the determining factor in choosing the appropriate model for estimating the average wire length.

In the process of further validating and extending the standard cell area estimation model we ran some test chip layouts on the MP2d layout system donated by RCA.

Higher-Level Estimation

We have investigated the area-speed tradeoffs exhibited by various RT-level constructs, such as adders, multipliers and the like. Initial observations suggest that the area-speed tradeoff curves tend to fit to curves of the type $AT^{\alpha}=k$ where A and T are area and time of the construct, respectively, k is a constant and α is an exponent dependent on the type of construct and its bit width.

We have investigated the possibility of estimating the cost/performance tradeoff curve of pipelined designs from the behavioral description.

In [3] we give a model for predicting cost-speed tradeoffs for pipelined designs. The model includes prediction of number of operators and registers from a behavior specification. It has been verified through the designs generated by the automated pipeline synthesis program Sehwa.

3.2. Synthesis

The synthesis work on this contract began with a study of the relationship between synthesis and verification. Further synthesis research has involved clocking scheme synthesis, pipelined and non-pipelined data path synthesis and register allocation [17].

A General Methodology

In [15] The general relationship between register-transfer synthesis and verification is discussed and common mechanisms are shown to underline both tasks. The paper proposes a framework for combined synthesis and verification of hardware ıat supports any combination of user-selectable synthesis techniques. The synthesis process can begin with any degree of completion of a partial design, and verification of the partial design can be achieved by completing its synthesis while subjecting it to constraints that can be generated from a "template" and user constraints. The driving force was the work done by Hafer [2] on a synthesis model. The model was augmented by adding variables and constraints in order to verify interconnections. A multilevel, multidimensional design representation [4, 1]is introduced which is shown to to be equivalent to Hafer's model. This equivalence relationship is exploited in deriving constraints off the design representation. These constraints can be manipulated in a variety of ways before being input to a linear program which completes the synthesis/verification process. An example is presented in which verification and synthesis occur simultaneously and the contribution of each automatically varies, depending on the number of previous design decisions.

The software illustrates the combined and verification of register-transfer designs. Mixed integer-linear constraints are derived from partial designs and then solved using mathematical programming. The missing design information is synthesized and design information already present is verified. This technique is facilitated by the chosen representation of the design information, which has an equivalence relationship with the mathematical program variables.

Clocking Scheme Synthesis

A theory of clocking was developed [8], [9], [13] and used as the basis for software which automatically synthesizes clocking schemes. This software synthesizes clocking schemes, given a partially complete register-transfer design. It determines the number and length of clock phases, how pipelining is to occur, and how many registers (stage latches) are required in the design in order for the clocking to work. The technique currently works when there is no resource sharing during a major cycle of the clock, and is being extended to exclude this limitation. The technique produces a 60% speed-up for a Hewlett-Packard 21MX computer when compared to the original design. The potential for application of the clocking scheme synthesis to more general systems, including systolic arrays, is also described, and an example given.

The algorithms and implementation for clocking scheme synthesis handle large problems. Currently, optimal clocking scheme synthesis for designs with about 120 modules and 300 interconnection nets takes 15 CPU minutes on a VAX/750. Minimization of the number of stage latches (in terms of total bitwidth) has been studied. The complexity of this problem when no degradation in performance is allowed is the same as that of the general assignment problem, and is NP-Complete. Several good heuristic solution techniques are being developed and compared. The program CSSP has been documented and is available on tape to the public for a minimal handling fee of $30, including documentation. A copy of the user's guide is attached.

A technique for automatic synthesis of clocking schemes for pipelined digital hardware has also been developed [10], [9], [11]. Two steps in the synthesis process are considered: the determination of number and location of pipeline partitions (stages) and the insertion of delays in the pipe in order to achieve minimum throughput latency without resource conflicts. We focus on a single reused resource in the same cycle, although extension of the solution techniques to cover multiple reused resources is straightforward.

Software exists to partition the system into stages subject to the number of stages or the maximum stage time, and to insert delays into the pipeline.

The delay insertion algorithm has been proven to be optimal. Although the algorithm performs exhaustive search, run times for large (20 stages) pipelines are less than a minute on a VAX 11/750.

Pipelined Data Path Synthesis

Synthesis of pipelined data paths has been investigated [10], [9], [14] [12]. This synthesis task involves the generation of data paths along with a clocking scheme which overlaps execution of multiple computation tasks. A theory of general execution overlap has been produced including four different scheduling techniques. We have produced a set of techniques for the synthesis of pipelined data paths, and written Sehwa, a program which performs such synthesis. The task includes the generation of data flow graph along with a clocking scheme which overlaps execution of multiple tasks. Some examples which Sehwa has designed are given in [3]. Sehwa can find the minimum cost design, the highest performance design, and other designs between these two in the design space. We believe Sehwa to be the first pipelined synthesis program published in the open literature.

The theory and technique for pipeline synthesis have been extended so that conditional branches can be handled. By handling conditionals, the technique can synthesize more sophisticated pipelines which can execute more than one type of task. The technique shares resources in an efficient manner, and thus produces cost-effective pipelines. This makes it possible to use a single pipeline for multiple types of tasks instead of either a complex reconfigurable pipeline or expensive multiple pipelines. This new extended technique has been added to the existing pipeline synthesis program called "Sehwa".

Sehwa is written in franz LISP, and executes within minutes for problems of practical size on a VAX 11/750. Documentation for Sehwa is also attached in the appendix, and Sehwa is available on tape.

Non-Pipelined Data Path Synthesis

A new RT-level non-pipelined datapath synthesis technique has been developed and programmed in Franz Lisp [16], and example datapaths synthesized. The program (MAHA) takes a data flow graph and a set of modules as input. The algorithm used is based on a linear module assignment to critical path operations, followed by a cost-based assignment using the concept of the "freedom". The freedom is a measure of tightness of the time limit for the whole input data flow. Operations with the least freedom are scheduled first. The program either minimizes cost subject to a time constraint, or maximizes speed, subject to a cost constraint.

MAHA is written in Franz LISP, and rewritten in C, and executes within minutes for problems of practical size on a VAX 11/750. MAHA documentation is attached, and MAHA is available on tape.

Register Allocation

The REAL REgister ALlocation program use a track assignment algorithm taken from channel routing called the Left Edge algorithm. REAL is optimal for non-pipelined designs with no conditional branches. It is thought that REAL is also optimal for designs with conditional branches, pipelined or not. Experimental results are included in [7], which illustrate the optimal solution found by REAL. REAL will be used to process designs output from MAHA and Sehwa. A summary of this research was described in [17].

3.3. Application of Design Representations

A VLSI design representation called the Design Data structure (DDS) has been developed at USC as part of the USC ADAM (Advanced Design AutoMation) project [4], [1]. The data structure based on this representation is implementation-independent and can be regarded as a general hardware design representation schema. It is characterized by four nonisomorphic hierarchies, which collectively describe the system under design. It has been used for a number of synthesis and analysis tasks including Sehwa and MAHA. Its requirements are being analyzed to determine the design and/or selection of appropriate user interfaces, including one or more hardware descriptive languages.

A working prototype of a program called *Catalog* has been constructed. As a part of the Advanced Design AutoMation (ADAM) system under development at USC, Catalog provides a format for storing and a method of accessing information about cell libraries and their contents using the DDS. Catalog provides a user-friendly interface between the database and other programs and will include a goal-driven macro-cell constructor called *Librarian* which combines cells from the selected library to form higher level cells. At the top level, Catalog is capable of guiding the user in the selection of a proper library. Catalog also provides detailed information concerning the data flow behavior, logical structure, physical details, and timing of each cell in the catalog in a format readable by a user or usable by a program. Work on *Librarian* and an interface with an object-oriented semantic data base which has been constructed under a separate contract is continuing.

4. PERSONNEL SUPPORTED

Alice C. Parker was supported as principal investigator 9/83 - 8/86.

Sarma Sastry was supported as a research assistant 9/83 - 12/84. Dr. Sastry was awarded a Ph.D. degree Jan. 1985, and is an Assistant Professor at USC.

Fadi Kurdahi was supported as a research assistant 9/83 - 8/86. Mr. Kurdahi expects to graduate June 1986.

Nohbyung Park was supported as a research assistant 9/83 - 12/85 and as a Postdoctoral Research Associate 1/86 - 6/86. Dr. Park received his Ph.D. degree Dec. 1985, and is an Assistant Professor at University of California, Irvine.

David Knapp was supported as a research assistant 9/83 - 6/84. David Knapp received his Ph.D. degree Dec. 1986, and is an Assistant Professor at the University of Illinois.

Jorge Pizarro was supported as a research assistant 1/86 - 6/86.

References

[1] Granacki, J.
Understanding Digital System Specifications Written in Natural Language.
PhD thesis, Dept. of Electrical Engineering - Systems, University of Southern California, December, 1986.

[2] Hafer, L., and Parker, A.
A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic.
IEEE Transactions on Computer-Aided Design CAD-2(1), January, 1983.

[3] Jain, R., Parker, A.C., and Park, N.
Predicting Area-Time Tradeoffs for Pipelined Design.
submitted to The 1987 Design Automation Conference.

[4] Knapp, D. and Parker, A.
A Unified Represention for Design Information.
In *Proceedings of the IFIP Conference on Hardware Description Languages.* IFIP, August, 1985.

[5] Kurdahi, F. and Parker, A.
Area Estimation of Standard Cell Designs.
Technical Report DISC-84-2, CRI-85-05, EE-Systems Dept. USC, 1985.

[6] Kurdahi, F. and Parker, A.
PLEST: A Program for Area Estimation of VLSI Integrated Circuits.
In *Proc. 23rd Design Automation Conf.*, pages 467-473. IEEE and ACM, June, 1986.

[7] Kurdahi, F., and Parker, A.
REAL: A Program for Register Allocation.
November, 1986.
Submitted to the 1987 Design Automation Conference.

[8] Park, N. and Parker, A.
Synthesis of Optimal Clocking Schemes for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

[9] Park, N.
Synthesis of High-Speed Digital Systems.
PhD thesis, Dept. of Electrical Engineering, University of Southern California. September, 1985.

[10] Park, N. and Parker, A.
Synthesis of Optimal Pipeline Clocking Schemes.
Technical Report DISC/85-1, Dept. of EE-Systems, University of Southern California, January, 1985.

[11] Park, N. and Parker, A.C.
Synthesis of Optimal Clocking Schemes.
In *Proceedings of the 22nd Design Automation Conference*, pages 489-495. IEEE and ACM, June, 1985.

[12] Park, N. and Parker, A.
Sehwa: A Program for Synthesis of Pipelines.
In *Proc. 23rd Design Automation Conf.*, pages 454-460. IEEE and ACM, June, 1986.

[13] Park, N. and Parker, A.C.
Theory of Clocking for Maximum Execution Overlap of High-Speed Digital Systems.
submitted to IEEE Transactions on Computers.

[14] Park, N. and Parker, A.C.
Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications.
submitted to IEEE Transactions on Computer-Aided Design.

[15] Parker, A., Kurdahi, F. and Mlinar, M.
A General Methodology for Synthesis and Verification of Register Transfer designs.
In *Proceedings of the 21st Design Automation Conference.* ACM SIGDA, IEEE Computer Society, June, 1984.

[16] Parker, A.C., Pizarro, J. and Mlinar, M.
MAHA: A Program for Datapath Synthesis.
In *Proc. 23rd Design Automation Conf.*, pages 461-466. IEEE and ACM, June, 1986.

[17] Parker, A.C., and Hayati, S.
Automating the VLSI Design Process.
accepted for publication in Proceedings of the IEEE.

[18] Sastry, S.
On the Relation between Wire Length Distributions and Placement of Logic on Master Slice ICs.
Technical Report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October, 1983.

[19] Sastry, S.
Wireability Analysis of Integrated Circuits.
PhD thesis, University of Southern California, 1984.

[20] Sastry, S. and Parker, A. C.
On the relation between wire length distributions and placement of logic on Master Slice ICs.
In *Proceedings of the 21st Design Automation Conference.* June, 1984.

Appendix A

A Guide To

CSSP

(Clocking Scheme Synthesis Package)

Version 2.1

By
Nohbyung Park
July 1986

Please direct inquiries to :

Dr. Alice Parker
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781
Arpanet address : parker@usc-cse.usc.edu

Table of Contents

List of Figures

1. INTRODUCTION

This document contains a brief introduction to the CSSP (Clocking Scheme Synthesis Package) which implements the clocking scheme synthesis algorithms developed by Nohbyung Park [Park 84, Park 85a,b]. We assume that the readers are familiar with the clocking scheme synthesis algorithms described in [Park 84, Park 85a,b].

Section 2 describes the input requirements of the system. Section 3 describes the operations of major routines of the system and how to use them. In Section 4, the global variables containing the current results of the clocking scheme synthesis and critical path analysis are described. Section 5 shows several example runs of this package.

2. INPUT FORMAT

The input to the CSSP consists of one or more directed acyclic graphs each of which is a Microcycle Execution Graph. The description of the input MEGs (Microcycle Execution Graphs) to be analyzed must be stored in a file. The input file must contain two lists, a node-set list and an edge-set list. Section 2.1 describes the formats for these lists. In Section 2.3, the types and usage of the pre-placed registers are described. In Section 2.3, two example of MEGs and their node and edge lists are given. Besides the node and edge description, the user can specify the default delay times, Dss and Dsp [Hafer 83], of the stage latches to be used. If not explicitly specified, they are set to zero.

2.1. Input File Format

An input file contains an ordered set of two LISP lists, <node-set> and <edge-set>.

```
file ::= <node-set> <edge-set>
```

Node-set contains the description of the functional modules in the input micro-cycle graph(s). Edge-set contains the description of the interconnections between the nodes in the input micro-cycle graph(s). If the input file format is not correct, the system will ask for a new file name. Any file can be edited using the *edit* command (refer to Section 3).

2.1.1. Node-set List

Node-set is a list of node-description lists. A node description list consists of the name of the corresponding functional module and its worst-case propagation delay. A node-set must contain at least two nodes.

```
<node-set> ::= (<node> <node> {<node>}*)

        <node> ::= (<nodename> <mpd>)

                <nodename> ::= a string of alphanumerics
                <mpd> ::= number
```

The nodename of each node must be unique and can be up to 80 characters long. *mpd* is the worst case (longest) delay time of the corresponding functional module. *mpd* can be zero if the node does not represent a real operational module (e.g. a dummy root node).

2.1.2. Edge-set List

An edge-set is a list of edge-description lists. An edge description is a list of up to seven-tuple of edge name, source node name, sink node name, bitwidth (bw), the number of delay registers (dr), the number of bypass registers (br), and the number of pre-placed stage latches (sl). The first four fields are mandatory. The types and usage of the registers will be discussed in Section 2.2.

```
<edge-set> ::= (<edge> {<edge>}*)

        <edge> ::= (<edgename> <source> <sink> <bw> [<dr> [<br> [sl]]])

                <edgename> ::= a string of alphanumerics
                <source> ::= <nodename>
                <sink> ::= <nodename>
                <bw> ::= integer
                <dr> ::= integer
                <br> ::= integer
                <sl> ::= integer
```

Edge names must be unique. Each edge must have one source and one sink node. Therefore, a root node with zero mpd must be used for all the input edges, and a terminal node with zero mpd for the output edges.

2.2. Pre-Placed Registers

Pre-palced registers are classified into three types:

1. **Delay registers**: registers which are used either to read or to write but *not to write and read* during one micro cycle. Mostly used to synchronize input/output data flow. These registers are carriers for values from microcycle to microcycle.

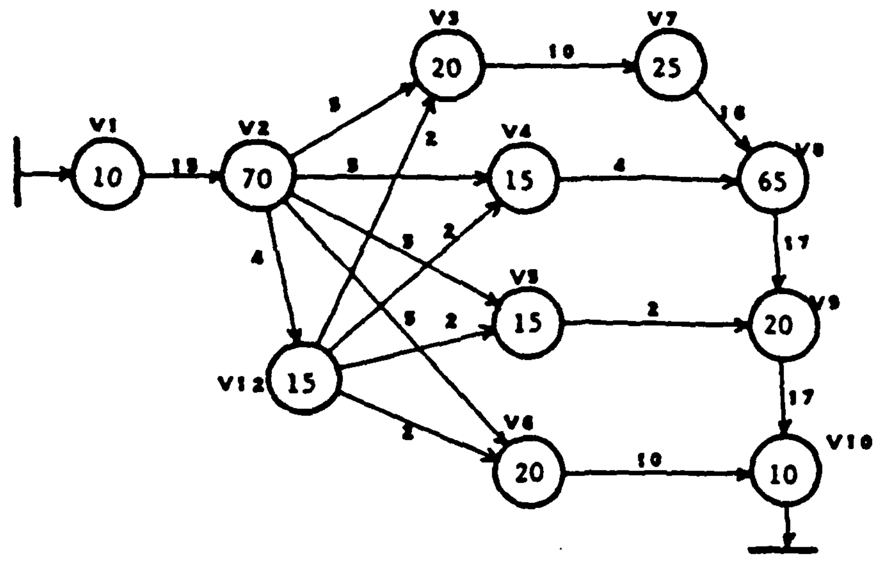
   Example: Registers in a systolic array which are clocked all at once.

2. **Bypass registers**: registers which are written and then read during one micro cycle. These registers do not affect the stage partitioning except the fact that each of them causes additional time delay equal to Dss + Dsp.

3. **Pre-placed stage latches**: registers which are pre-placed to force certain edges to be included in a cutset.

If the corresponding fields of these registers in an edge description are unspecified they are set to zero.

2.3. Examples

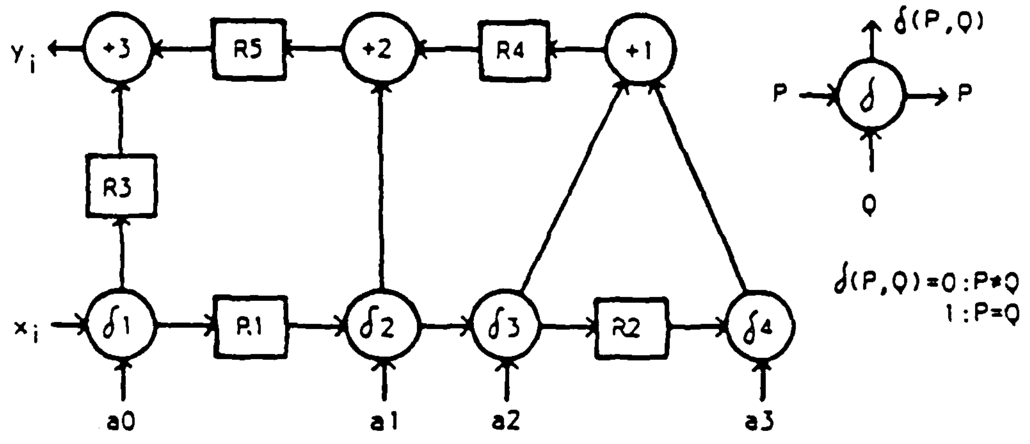


```
;21MX-E CPU Non-Branch Group Instructions


;Node List
( (v1 10) (v2 70) (v3 20) (v4 15) (v5 15) (v6 20)
  (v7 25) (v8 65) (v9 20) (v10 10) (v12 15)  )


;Edge List
( (PA v1 v2 15 0 0)        (IF1 v2 v3 5 0 0)
  (IF2 v2 v4 5 0 0)        (IF3 v2 v5 5 0 0)
  (IF4 v2 v6 5 0 0)        (OP v2 v12 4 0 0)
  (SRC v12 v3 2 0 0)       (ALU v12 v4 2 0 0)
  (SR v12 v5 2 0 0)        (DST v12 v6 2 0 0)
  (CS1 v3 v7 10 0 0)       (CS2 v4 v8 4 0 0)
  (CS3 v5 v9 2 0 0)        (CS4 v6 v10 10 0 0)
  (INPUT v7 v8 16 0 0)     (RESULT v8 v9 17 0 0)
  (OUTPUT v9 v10 16 0 0) )
```

Figure 2-1: Example 1



A systolic array evaluating $\sum_{j=0}^{3} \delta(x_{i-j},a_j)$.

```
; Systolic Array for Convolution


; node list
(
(delta1 3) (delta2 3) (delta3 3) (delta4 3)
(add1 7) (add2 7) (add3 7)
)


;edge list
(
(e0 delta1 delta2 1 1 0)        (e1 delta2 delta3 1 0 0)
(e2 delta3 delta4 1 1 0)        (e3 delta1 add3 1 1 0)
(e4 delta2 add6 1 0 0)          (e5 delta3 add1 1 0 0)
(e6 delta4 add1 1 0 0)          (e7 add1 add2 1 1 0)
(e8 add2 add3 1 1 0)
)
```

Figure 2-2: Example 2

3. BASIC OPERATIONS OF THE CSSP

The CSSP consists of four major procedures, init, kpart, opart and cp. These main synthesis procedures together with other utility routines are interfaced to a user through a command line interpreter.

For more details of these procedures, the reader is urged to refer to the source code in the Appendix.

init [filename]

This procedure initializes the CSSP with a new design to be analyzed. *Filename* is the input file containing the description of the MEG(s) to be analyzed. This procedure reads in the input graph and sets up necessary data structures. For the input file format, refer to Section 2.

set

The user can set the register set-up time, Dss, and the propagation delay, Dsp. Default values are zeros.

kpart [stage-time limit]

The procedure *kpart* partitions the MEG(s) into the minimum number of stages each of which has stage propagation delay no longer than *stage-time limit*. The resulting edge cutsets, the number of stages, and stage propagation delays are returned.

opart [k]

The procedure **opart** partitions the MEG(s) into exactly **k** partitions whenever it is possible (there must be at least one directed path with more than k nodes with non-zero propagation delays in the MEG(s)). If there are more than one such partitions, opart chooses one with shortest maximum-stage-propagation-delay. The results returned are the same as that of the *kpart* procedure.

cp

The procedure **cp** finds the critical path(s) in the MEG(s). The procedure cp uses actual module propagation delays to compute the critical path delay. There can be more than one critical path. The outputs from the procedure cp are lists of nodes and edges on the critical paths and the lengths of the critical path(s).

exam

User can access the values of the atoms that are global variables. Refer to Section 4 for the description of global atom variables.

edit [file name]

Edit a file using **emacs.**

load [file name]

Load a file.

exit

Exit from the CSSP. On exit, the system asks whether the user wants to analyze the results. The analyzed results are written out to a file *cssp.log.*

4. GLOBAL ATOM VARIABLES

The global atom variables can be examined using the **exam** command at any stage of the execution of the CSSP. Especially, when there is run-time error, these variables can be used to trace the cause of the error.

intervals	A list of all the possible stage times. This is an enumeration of the longest path delays from each node to every node.
dss	Set-up time of the stage latches.
dsp	Propagation delay of the stage latches.
edgelist	Edge-set list.
nodelist	Node-set list.
numofnodes	The number of nodes.
mpd	A list of module propagation delays of the nodes.
numofedges	The number of edges.
BWList	A list of the bitwidth of the edges.
outvalues	A list of output edges.
invalues	A list of input edges.
lmax	Current maximum stage-time limit.
k	The number of stages as the result of the last stage partitioning.
eh	Current set of edges being traversed during stage partitioning.
nh	Current set of nodes being visited during stage partitioning.

5. EXAMPLES

Example 1: The MEG of Figure 1

```
-> (cssp)

Welcome to CSSP!
Use <init [filename]> command to read initial micro-cycle graph(s).
Type <help> or <help init> for help.

CSSP> init ex1

*****  MODULE LIST  *****

   (MODULE_NAME MPD<nsec>)

   m 0  =  (v1 10)
   m 1  =  (v2 70)
   m 2  =  (v3 20)
   m 3  =  (v4 15)
   m 4  =  (v5 15)
   m 5  =  (v6 20)
   m 6  =  (v7 25)
   m 7  =  (v8 65)
   m 8  =  (v9 20)
   m 9  =  (v10 10)
   m 10 =  (v12 15)

*****  EDGE LIST  *****

(EDGE_NAME SRC SINK BW DR BR SL)

   e 0  =(PA v1 v2 15 0 0)
   e 1  =(IF1 v2 v3 5 0 0)
   e 2  =(IF2 v2 v4 5 0 0)
   e 3  =(IF3 v2 v5 5 0 0)
   e 4  =(IF4 v2 v6 5 0 0)
   e 5  =(OP v2 v12 4 0 0)
   e 6  =(SRC v12 v3 2 0 0)
   e 7  =(ALU v12 v4 2 0 0)
   e 8  =(SR v12 v5 2 0 0)
   e 9  =(DST v12 v6 2 0 0)
   e 10 =(CS1 v3 v7 10 0 0)
   e 11 =(CS2 v4 v8 4 0 0)
   e 12 =(CS3 v5 v9 2 0 0)
```

```
    e 13  =(CS4 v6 v10 10 0 0)
    e 14  =(INPUT v7 v8 16 0 0)
    e 15  =(RESULT v8 v9 17 0 0)
    e 16  =(OUTPUT v9 v10 16 0 0)

CSSP> set
Dss? 5
Dsp? 10

CSSP> kpart 100

 **********  EDGE-CUTSETs  **********
  cutset  1  =(4 9 3 8 2 7 1 6)
  cutset  2  =(12 13 11 14)
  cutset  3  =(13 16)

 *****  STAGE-PROPAGATION-DELAY  *****
    stage-delay  1  =100
    stage-delay  2  =60
    stage-delay  3  =100
    stage-delay  4  =20
Total Processing Time: 1 seconds  32 (1/60 sec.)

CSSP> cp

CRITICAL_PATH_NODES
0 1 2 6 7 8 9 10

CRITICAL_PATH_EDGES
0 5 6 10 14 15 16
Total Processing Time: 1 seconds  20 (1/60 sec.)

CSSP> opart 3

Realizable Stage Times are:
(85 95 100 105 115 120 125 130 140 145 150 160 165 210 230 235)

Enter kpart with stage time: 130  --->  k =  2
Enter kpart with stage time: 105  --->  k =  3
Enter kpart with stage time:  95  --->  k =  4
Enter kpart with stage time: 100  --->  k =  4

"Enter kpart with stage time:" 105

 **********  EDGE-CUTSETs  **********
  cutset  1  =(4 9 3 8 2 7 1 6)
  cutset  2  =(12 13 11 14)

 *****  STAGE-PROPAGATION-DELAY  *****
    stage-delay  1  =100
    stage-delay  2  =60
    stage-delay  3  =105
Total Processing Time: 1 seconds  32 (1/60 sec.)

dmax = 105
Total Processing Time: 11 seconds  9 (1/60 sec.)

CSSP> exit
Do you want to analyze the results? y

*** Edge Cutsets ***
 Stage    1: IF4 DST IF3 SR IF2 ALU IF1 SRC
 Stage    2: CS3 CS4 CS2 INPUT

*** Stage Propagation Delays ***
 Stage    1: 100
 Stage    2: 60
 Stage    3: 105

Minimum Clock Period (tcy) = 105
Number of Clock Phases (Max.) = 3
Optimal Clock Cycle:
   When resync. rate >= 0.355 : 132.5
   When resync. rate <  0.355  : 105
BYE.
t
```

Example 2: The MEG of Figure 2.

```
-> (cssp)

Welcome to CSSP!
Use <init [filename]> command to read initial micro-cycle graph(s).
Type <help> or <help init> for help.

CSSP> init leis

*****  MODULE LIST  *****

    (MODULE_NAME MPD<nsec>)
    m 0 = (v1 3)
    m 1 = (v2 3)
    m 2 = (v3 3)
    m 3 = (v4 3)
    m 4 = (v5 7)
    m 5 = (v6 7)
    m 6 = (v7 7)

***** EDGE LIST *****

(EDGE_NAME SRC SINK BW DR ER SL)

    e 0 =(e0 v1 v2 1 1 0)
    e 1 =(e1 v2 v3 1 0 0)
    e 2 =(e2 v3 v4 1 1 0)
    e 3 =(e3 v1 v7 1 1 0)
    e 4 =(e4 v2 v6 1 0 0)
    e 5 =(e5 v3 v5 1 0 0)
    e 6 =(e6 v4 v5 1 0 0)
    e 7 =(e7 v5 v6 1 1 0)
    e 8 =(e8 v6 v7 1 1 0)

CSSP> set
Dss? 5
Dsp? 5

CSSP> set
Dss? 1
Dsp? 1

CSSP> kpart 8

 ********** EDGE-CUTSETs **********
  cutset 1 =(4 3 5 6)

 ***** STAGE-PROPAGATION-DELAY *****
    stage-delay 1 =8
    stage-delay 2 =8
Total Processing Time: 57 (1/60 sec.)

CSSP> opart 2

Realizable Stage Times are:
(8 11 14)

Enter kpart with stage time:  11  --> k = 2
Enter kpart with stage time:   8  ——>  k =  2

"Enter kpart with stage time:"   8

 **********  EDGE-CUTSETs  **********
  cutset  1  =(4 3 5 6)

 *****  STAGE-PROPAGATION-DELAY  *****
    stage-delay  1  =8
    stage-delay  2  =8
Total Processing Time: 43 (1/60 sec.)

dmax =   8
Total Processing Time: 3 seconds  1 (1/60 sec.)

CSSP> exit
Do you want to analyze the results? y

*** Edge Cutsets ***
 Stage   1: e4 e3 e5 e6

*** Stage Propagation Delays ***
 Stage   1: 8
 Stage   2: 8

Minimum Clock Period (tcy) = 8
Number of Clock Phases (Max.) = 2
Optimal Clock Cycle = 8
BYE.
t
->
```

REFERENCES

[Park 84]	Park, N. and Parker, A. *Synthesis of Optimal Clocking Schemes for Digital Systems.* Technical Report DISC/84-1, Dept. of EE-Systems University of Southern California, May 1984.
[Park 85a]	Park, N. and Parker, A. Synthesis of Optimal Clocking Schemes. In *Proceedings of the 22nd Design Automation Conference* ACM and IEEE, June 1985.
[Park 85b]	Park, N. *Synthesis of High-Speed Digital Systems.* PhD Thesis, Dept. of Electrical Engineering-Systems University of Southern California, September 1985.
[Hafer 83]	Hafer, L. and Parker, A. A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic. IEEE Transactions on Computer-Aided Design, CAD-2(1), January 1983.
[Leiserson 83]	Leiserson, C. E., Rose, F. M. and Saxe, J. B. Optimizing Synchronous Circuitry by Retiming. In *Proceedings of Third Caltech Conference on VLSI* Computer Science Press, 1983.

APPENDIX

CSSP

Program Listings

Appendix II

Sehwa
USER'S MANUAL

A Pipeline Synthesis Program

The ADAM - Advanced Design AutoMation Project

University of Southern California

by

Rajiv Jain

Please direct inquiries to :

Dr. Alice Parker, Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, CA 90089-0781. Arpanet address : *parker@usc -cse.usc.edu* .

INTRODUCTION

Sehwa is a program which synthesizes pipelined data paths from an input dataflow graph. Sehwa performs allocation of functional modules and scheduling of resources. Sehwa estimates the cost of registers and interconnects, but does not perform the detailed allocation of these elements.

Sehwa is general purpose, in that it can take into account data dependencies, conditional branching in the input specification and the resynchronization rate due to exceptions. Sehwa finds the fastest design, the cheapest design and a range of optimal designs in between these two extremes.

Sehwa was written in Franz LISP by Dr. Nohbyung Park as a part of his Ph.D. dissertation [1]. It runs on a Sun workstation as well as Vax 11/750.

This document is Sehwa user's manual. It is not intended to be a treatise on pipelined synthesis ([1] is a good reference for this). It is assumed that the user knows the capabilities of the program. It is also assumed that the user has experience of working on the BSD Unix operating system and an editor. Although the understanding of this document does not require it, the user is cautioned that the use of Sehwa requires the abovementioned skills. The document is divided into three sections : data preperation, how to run Sehwa and finally an example with results.

DATA PREPERATION

The input to Sehwa is a data flow graph in Park Normal Form [1]. PNF has a LISP like structure and consists of two lists : the *node* list and the *edge* list. They are

(nodes (*node_ 1_description*) (*node_ 2_description*) .. (*node_n_description*))

(edges (*edge_ 1_description*) (*edge_ 2_description*) .. (*edge_n_description*))

The bold words are terminal and the words in italics are non-terminals.

The node description is specified as follows :

node_name node_type bit_width

where, *node_name* is the instantiation of the node, *node_type* is the operation performed by the node and *bit_width* is the number of bits (in integer) handled by the node. An example of node list is

(nodes (a1 add 16) (m1 mult 16)).

The edges are similarly specified as :

edge_name source_node destination_node bit_width value_name

where, *edge_name* is the name of the edge, *source_node* is the node it starts from (if the edge is an input edge from external the *source_node* is *root*), *destination_node* is the node the edge connects to (in the case of the edge carrying a value external to the data flow graph the *destination_node* is *outport*) and *value_name* is the value carried by the edge. The *value_name* is especially useful where two (or more) edges emanating from the same node carry the same value. An example edge list is

(edges (e1 root a1 16 x) (e2 a1 m1 16 e2) (e3 m1 outport 16 y)).

For conditional branching Sehwa has two reserved words *dist* and *join* . A node is specified as one of these as

(*node_name* dist)

(*node_name* join).

Each *dist* node must pair up with a *join* node in the dataflow graph.

A complete list of reserved words recognized by Sehwa in the input file is : *nodes* , *edges* , *root* , *outport* , *dist* and, *join* .

USING Sehwa

Having prepared the input file containing the data flow graph in PNF, we now describe the use of Sehwa. As Sehwa is written in LISP, the user has to first enter the LISP environment by typing

```
lisp*
```

The next step is to load the file contaning Sehwa source code and invoke it. This is achieved through

```
[load 'Sehwa]
[Sehwa]
```

Note the *quote* before Sehwa in the first command and the upper case S. On invocation, Sehwa will prompt for the input file and then for the information about the set of modules used for implementing the nodes. The module information is specified as

module_name operation bit_width area delay

where, *module_name* is the name of the module, *operation* is the function performed by the node, and *delay* is the module delay (it has to be an integer). An example of an 8 bit carry save adder description is

```
csadder add 8 0.5 30
```

Sehwa then prompts the user for latch information. The latch information is required to determine stage delays and in making estimate of register costs. The following information is requested by Sehwa :

- Setup time of the latch
- Propagation time through the latch, and
- Area per bit of the latch.

* A carriage return is implicit after each command.

Sehwa then proceeds to compute the boundary design points. At this point Sehwa has all the information and the user can now proceed to get results by specifying his cost/speed constraint. The constraint is specified by typing in order

cost (or) speed

constraint value

resynchronization value.

To exit the program type *exit*. Sehwa will then save the previous constraint results in a file *sehwa.log* and exit to the LISP environment.

Known bugs : There are situations which the user may come across while executing Sehwa and due to the lack of suitable title they have been called bugs. These are :

1. At some point before the main synthesis loop, the user might wish to exit Sehwa. The only effective way of doing that is by typing ^C (control-C) and going to the LISP environment. There is no graceful way of doing this.

2. When Sehwa prompts for the first time for module descriptions, the program goes into an infinite loop if the user types a *carriage return* only. The user then has to type ^C to break the infinite loop.

EXAMPLE and RESULTS

An example dataflow graph using conditional branches is shown in Fig. 1. The input file in PNF and a sample run for the same is included. User typed words are underlined. The results give a detailed account of the scheduling of operations, the number of resources required, the latency, an estimated cost of registers and the total estimated cost of the design. There can be situations in conditional branching when two operations are scheduled in the same time step to the same module. In this situation the operation nodes which are designated for the same resource are clustered together in angular brackets in the output.

REFERENCE

1. N. Park, "Synthesis of High-Speed Digital Systems", *Technical Report CRI-85-23, Computer Research Institute, University of Southern California.*

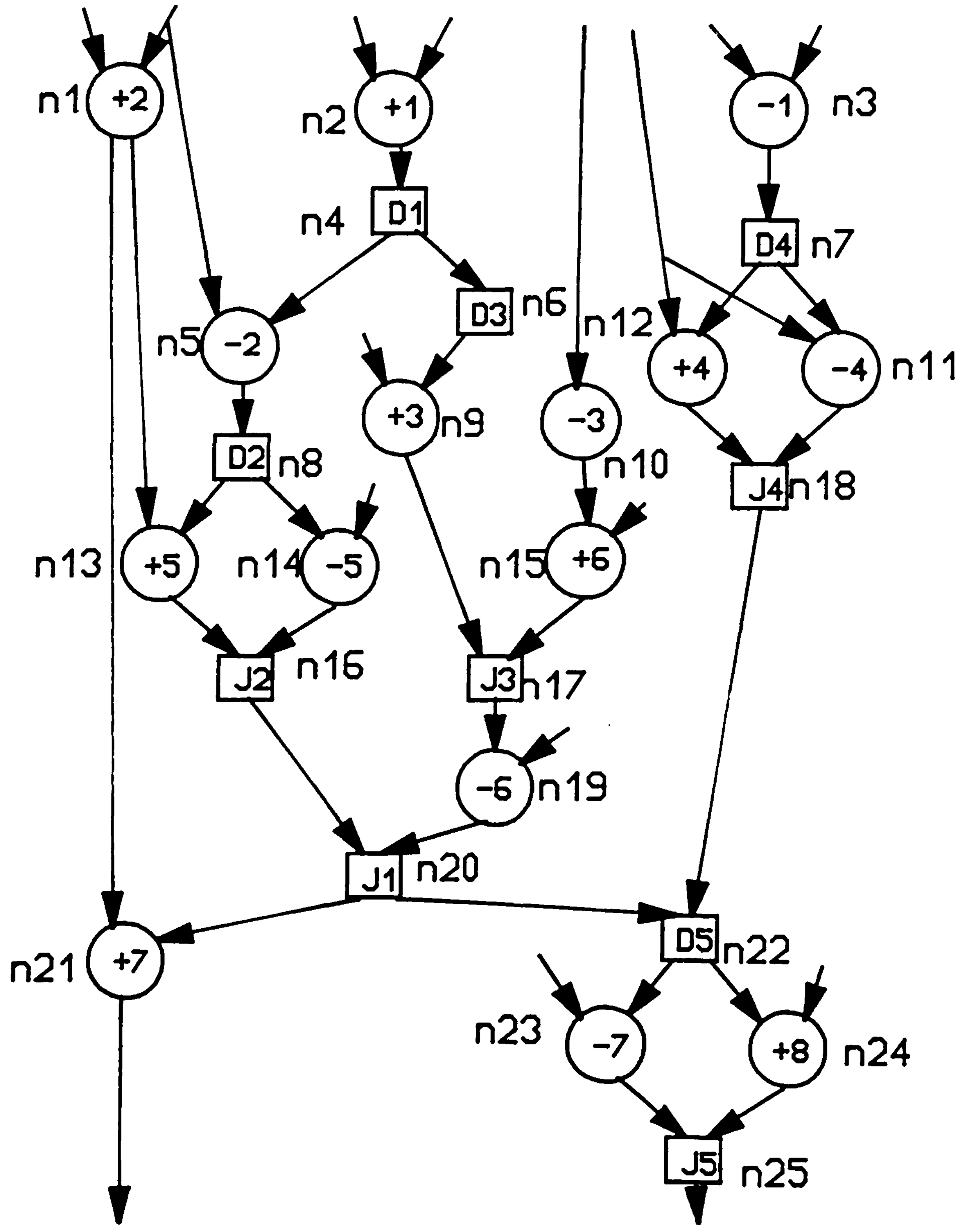


Fig. 1 Example Dataflow Graph

```
(nodes
    (n1 add 16)
    (n2 add 16)
    (n3 sub 16)
    (n4 dist)
    (n5 sub 16)
    (n6 dist)
    (n7 dist)
    (n8 dist)
    (n9 add 16)
    (n10 sub 16)
    (n11 sub 16)
    (n12 add 16)
    (n13 add 16)
    (n14 sub 16)
    (n15 add 16)
    (n16 join)
    (n17 join)
    (n18 join)
    (n19 sub 16)
    (n20 join)
    (n21 add 16)
    (n22 dist)
    (n23 sub 16)
    (n24 add 16)
    (n25 join)
)

(edges
    (v1 root n1 16 v1)
    (v2 root n1 16 v2)
    (v3 root n5 16 v2)
    (v4 root n2 16 v4)
    (v5 root n2 16 v5)
    (v6 root n10 16 v6)
    (v7 root n12 15 v7)
    (v8 root n3 16 v8)
    (v9 root n3 16 v9)
    (v10 n1 n21 16 v10)
    (v11 n1 n13 16 v11)
    (v12 n2 n4 16 v12)
    (v13 n3 n7 16 v13)
    (v14 n4 n5 16 v14)
    (v15 n4 n6 16 v15)
    (v16 root n11 16 v7)
    (v17 n7 n12 16 v17)
    (v18 n7 n11 16 v18)
    (v19 n5 n8 16 v19)
    (v20 root n9 16 v10)
    (v21 n6 n9 16 v21)
    (v22 n6 n10 16 v22)
    (v23 n8 n13 16 v23)
    (v24 n8 n14 16 v24)
    (v25 root n14 16 v25)
    (v26 n9 n17 16 n26)
    (v27 n10 n15 16 v27)
    (v28 root n15 16 v28)
    (v29 n12 n18 16 v29)
    (v30 n11 n18 16 v30)
```

```
        (v31 n13 n16 16 v31)
        (v32 n14 n16 16 v32)
        (v33 n15 n17 16 v33)
        (v34 n18 n22 16 v34)
        (v35 n16 n20 16 v35)
        (v36 n17 n19 16 v36)
        (v37 root n19 16 v37)
        (v38 n19 n20 16 v38)
        (v39 n20 n21 16 v39)
        (v40 n20 n22 16 v40)
        (v41 n21 outport 16 v41)
        (v42 n22 n23 16 v42)
        (v43 n22 n24 16 v43)
        (v44 n23 n25 16 v44)
        (v45 n24 n25 16 v45)
        (v46 n25 outport 16 v46)
        (v47 root n23 16 v47)
        (v48 root n24 16 v24)
    )
```

```
rajiv[1] >>lisp
Franz Lisp, Opus 38.79
-> [load 'Sehwa]
[load Sehwa.l]
t
-> [Sehwa]

        U   U   SSS    CCC        SSS   EEEEE  H    H   W    W    A
        U   U  S   S  C   C      S   S  E      H    H   W    W   A A
        U   U  S      C          S      E      H    H   W    W  A   A
        U   U   SSS   C     ===   SSS   EEEEE  HHHHH    W    W  A   A
        U   U      S  C              S  E      H    H   W W W   AAAAA
        U   U  S   S  C   C      S   S  E      H    H   W W W   A   A
         UUU    SSS    CCC        SSS   EEEEE  H    H    W W    A   A


          WELCOME TO USC-SEHWA PIPELINE SYNTHESIS PACKAGE!

                *** USC Design Automation Group ***
                        Nohbyung  Park


; ************************************* ;
;               PHASE 1                 ;
;                                       ;
;         Input Processing and          ;
;     Data Structure Initialization     ;
; ************************************* ;

Input Data-Flow Graph (File Name)? pk.l

Global buffer matrices allocated..

Data flow graph translated..


; ************************************* ;
;               PHASE 2                 ;
;                                       ;
;         Module Selection and          ;
;              Synthesis                ;
; ************************************* ;

Global data structure being constructed ..


 ***************************************
 *                                     *
 *   WELCOME TO MODULE-SELECTION PHASE *
 *                                     *
 ***************************************

Do you want instructions (y/n)? y

 <INSTRUCTION FOR MODULE SELECTION>

This is an interactive and iterative module-selection
```

```
routine.  Type-in a  module-description list for each
function whose name will be printed one by one.

MODULE DESCRIPTION FORMAT:
<module> ::= (<module_name><operation><bitwidth><cost><delay_time>)

Each module_name must be unique among module names.
The bitwidth and module delay time (nsec) are positive
integers and the cost is a positive real number.

Ex1:   csadder8 add 8 0.5 30  -- An 8-bit carry-save adder

Ex2:   TI74284 mul 4 2.0 60   -- TI 4-bit binary multiplier


Type carriage-return (CR) to continue ...


    *** Function List ***
 (function  (op-node indices))

(
   0  (add (14 12 10 8 7 4 1 0))
   1  (sub (2 3 5 6 9 11 13))
)


For each function, type a module description list.
(Type carriage-return (CR) to skip for no change.)

> Function:  add
  Total number of nodes: 8
  Max. number of possible evaluations: 6
  Previous Assignment: None
  (New) Module Description? a add 16 4200 340

> Function:  sub
  Total number of nodes: 7
  Max. number of possible evaluations: 5
  Previous Assignment: None
  (New) Module Description? s sub 16 4200 340


Module Selection is complete!
Do you want any change (y/n)? n


Stage Latch Information:
  Dss (set-up time in nsec.) ? 11
  Dsp (propagation time in nsec.) ? 5
  Unit cost (per bit) ? 15.624


Computing All Possible Stage Times ..


Design-space boundaries computation started ..

***************************************
*                                     *
*  Design-Space Boundary Information   *
*                                      *
****************************************


>> Fastest Design:

Nodes-to-stages Assignment:
   Stage 0:  0(n1) 1(n2) 2(n3) 15(n4) 16(n6)
             17(n7)
   Stage 1:  3(n5) 4(n9) 5(n10) 6(n11) 7(n12)
             18(n8) 22(n18)
   Stage 2:  8(n13) 9(n14) 10(n15) 20(n16) 21(n17)
   Stage 3:  11(n19) 19(n22) 23(n20)
   Stage 4:  12(n21) 13(n23) 14(n24) 24(n25)
Clock Cycle (initiation interval): 356
   Stage Time (minor clock cycle): 356
   Latency: 1
Effective Initiation Interval: 356
   Resynchronization rate: 0 %
Total Cost: 77467.82399999999
   Module cost: 63000
   # of modules:
      a: 8
      s: 7
   Latch cost : 14467.824
Scheduling Algorithm Used: Forward-Maximum-Scheduling


>> Cheapest design:

Nodes-to-stages Assignment:
   Stage 0:  18(n8) < 5(n10) 3(n5) > 16(n6) 15(n4) 1(n2)
   Stage 1:  17(n7) 2(n3) 0(n1)
   Stage 2:  22(n18) 6(n11) 7(n12)
   Stage 3:  19(n22) 23(n20) 20(n16) < 9(n14) 11(n19) > 21(n17)
             < 8(n13) 4(n9) 10(n15) >
   Stage 4:  24(n25) 13(n23) 14(n24)
   Stage 5:  12(n21)
Clock Cycle (initiation interval): 4176
   Stage Time (minor clock cycle): 696
   Latency: 6
Effective Initiation Interval: 4176
   Resynchronization rate: 0 %
Total Cost: 25852.008
   Module cost: 8400
   # of modules:
      a: 1
      s: 1
   Latch cost : 17452.008
Scheduling Algorithm Used: Forward-Feasible-Scheduling


>> Absolute Boundaries:

     Absolute Minimum Cost: 25852.008

     Absolute Minimum Initiation Interval: 356
```

```
****************************************
*                                      *
*   WELCOME!! to Main Synthesis Loop   *
*                                      *
****************************************


Select Optimization Mode (cost/speed/exit)? cost

Maximum Allowable Cost? 50000

Expected Resynchronization Rate (in %)? 0

Design in progress with:
  Latency: 1
  # of modules: (6 5)
  Stage times: (356 696 1036 1376 1716)

> Tentative Solutions Found

Nodes-to-stages Assignment:
   Stage 0:  17(n7) 0(n1) 2(n3) 16(n6) 15(n4)
             1(n2)
   Stage 1:  22(n18) 18(n8) 6(n11) 7(n12) < 3(n5) 5(n10) >
   Stage 2:  21(n17) < 8(n13) 4(n9) 10(n15) >
   Stage 3:  19(n22) 23(n20) 20(n16) < 11(n19) 9(n14) >
   Stage 4:  24(n25) 12(n21) 13(n23) 14(n24)
Clock Cycle (initiation interval): 356
   Stage Time (minor clock cycle): 356
   Latency: 1
Effective Initiation Interval: 356
   Resynchronization rate: 0 %
Total Cost: 61417.776
   Module cost: 46200
   # of modules:
      a: 6
      s: 5
   Latch cost : 15217.776
Scheduling Algorithm Used: Forward-Feasible-Scheduling



Design in progress with:
  Latency: 2
  # of modules: (3 3)
  Stage times: (356 696 1036 1376 1716)

> Tentative Solutions Found


Nodes-to-stages Assignment:
   Stage 0:  1(n2) 2(n3) 15(n4) 17(n7) 16(n6)
             0(n1) 7(n12) < 3(n5) 5(n10) >
   Stage 1:  18(n8) 6(n11) < 4(n9) 8(n13) 10(n15) > 22(n18) < 11(n19) 9(n14) >
             21(n17) 20(n16) 23(n20) 19(n22) 14(n24)
             13(n23) 12(n21) 24(n25)
Clock Cycle (initiation interval): 2072
   Stage Time (minor clock cycle): 1036
   Latency: 2
Effective Initiation Interval: 2072
```

```
   Resynchronization rate: 0 %
Total Cost: 32683.896
   Module cost: 25200
   # of modules:
      a: 3
      s: 3
   Latch cost : 7483.89600000001
Scheduling Algorithm Used: Backward-Feasible-Scheduling


        *** SOLUTION ***

Nodes-to-stages Assignment:
   Stage 0:  17(n7) 0(n1) 2(n3) 16(n6) 15(n4)
             1(n2)
   Stage 1:  22(n18) 18(n8) 6(n11) 7(n12) < 5(n10) 3(n5) >
   Stage 2:  20(n16) 21(n17) < 8(n13) 4(n9) 10(n15) > 9(n14)
   Stage 3:  19(n22) 23(n20) 11(n19)
   Stage 4:  13(n23)
   Stage 5:  24(n25) 14(n24) 12(n21)
Clock Cycle (initiation interval): 712
   Stage Time (minor clock cycle): 356
   Latency: 2
Effective Initiation Interval: 712
   Resynchronization rate: 0 %
Total Cost: 41167.728
   Module cost: 25200
   # of modules:
      a: 3
      s: 3
   Latch cost : 15967.728
Scheduling Algorithm Used: Forward-Feasible-Scheduling

 * Second Alternative:

Nodes-to-stages Assignment:
   Stage 0:  22(n18) 7(n12) 6(n11) 18(n8) 17(n7)
             < 3(n5) 5(n10) > 0(n1) 2(n3) 16(n6) 15(n4)
             1(n2)
   Stage 1:  19(n22) 23(n20) 20(n16) < 11(n19) 9(n14) > 21(n17)
             < 8(n13) 4(n9) 10(n15) >
   Stage 2:  24(n25) 12(n21) 13(n23) 14(n24)
Clock Cycle (initiation interval): 696
   Stage Time (minor clock cycle): 696
   Latency: 1
Effective Initiation Interval: 696
   Resynchronization rate: 0 %
Total Cost: 54933.816
   Module cost: 46200
   # of modules:
      a: 6
      s: 5
   Latch cost : 8733.81600000001
Scheduling Algorithm Used: Forward-Feasible-Scheduling


Select Optimization Mode (cost/speed/exit)? exit
Solution List is written out to <sehwa.log>.
```

```
          ***                        ***
      *         *                *         *
    *    O    O    *          *    O    O    *
    *       |      *          *       |      *
    *    \____/    *    OR    *               *      ??
      *         *                *   /____\   *
          ***                        ***
```

```
Bye ..
t
-> ^D
Goodbye
```

Appendix III

MAHA USER'S MANUAL

A Datapath Synthesis Program

The ADAM - Advanced Design AutoMation Project

University of Southern California

by

Mitchell Mlinar

Please direct inquiries to :

Dr. Alice Parker, Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, CA 90089-0781. Arpanet address : *parker@usc-cse.usc.edu* .

Table of Contents

1. Introduction

The ADAM system at the University of Southern California has as a goal the fabrication of hardware from a high-level description. One portion of that system includes datapath synthesis. **MAHA** is a non-pipelined datapath synthesis program which takes as its input a dataflow graph description. This graph is derived from the **DDS** (Design Data Structure) which is at the heart of the ADAM system.

This manual describes **MAHA** as well as a number of utilities which generate input data files for **MAHA**. One set of utilities converts from a **VT** (Value-Trace) description to **PNF** (Park Normal Form) - the data structure used by the MAHA program. Another set of utilities convert from the older LISP-format PNF data structures to the current text format used by the C version of **MAHA**.

2. MAHA

This section details the use of the MAHA prograr Items which will be discussed include the input data structures to **MAHA**, running **MAHA**, and the output file which may be generated by **MAHA**.

2.1 Introduction

MAHA is a program which takes as an input a dataflow graph, module library, and global constraints and outputs the allocation data which includes the timing, bindings, and cost of the synthesis. **MAHA** can work with the **PNF** description directly, although it can also work with any properly conditioned **VT** description.

Getting to **PNF** from a **VT** description is discussed in the next chapter. A detailed description of C functions and the internal data structure of **MAHA** are presented in Appendix I.

2.2 MAHA Inputs

Although **MAHA** relies on an interactive environment to control program flow, the dataflow graph cannot be entered at runtime. Two data input files are required by **MAHA**: a dataflow description and a module library. The dataflow description consists of a single file which contains both a list of nodes and a list of edges; the module library consists of modules and their associated cost, speed, and bit-width. Although both the dataflow file and library file may be named in any manner of your choosing, it is recommended that the primary names be identical while changing the extension to indicate the type of file. For example, the dataflow might be contained in *garbage.dfg* and the module library in *garbage.lib*.

The *dataflow description file* contains the description of a complete dataflow graph. The graph has a number of restrictions on it:

- The graph can contain no more than 1000 nodes or 2000 edges.
- The graph must be acyclic (no loops).

- The top of the graph is indicated by the reserved word **root** which cannot be defined elsewhere. **Root** has NO input edges, only output edges. If you do not have a **root** node, **MAHA** will add one for you along with the necessary edges to it. If you have a **root** node, it should be of type *dummy* since it does not physically represent any operation.

- The bottom of the graph is indicated by the reserved word **outport** and cannot be defined elsewhere. **Outport** has NO output edges, only input edges. If you do not have a **outport** node, **MAHA** will add one for you along with the necessary edges to it. If you have a **outport** node, it should be of type *dummy*.

- *Conditional branches* are indicated by the reserved words **dist** for fork (distribution) nodes and **join** for join nodes. **Dist** has one input arc and one or more output arcs; **join** is the converse.

- *Parallel branches* are indicated by the reserved words **parbeg** where the parallel branches begin and **parend** where the parallel branches recombine.

- In the case of multiple output edges from a given node, **MAHA** will attempt to treat them as parallel branches with an implied **parbeg**. The implied **parend** for this group is at **outport**.

The *dataflow description file* has both node and edge information in a one node (edge) per line format as follows:

```
node-description-1
node-description-2
..................
node-description-n

edge-description-1
edge-description-2
..................
edge-description-m
```

Note the blank line between the node and edge descriptions; this is a REQUIREMENT.

2.2.1 Node Description

A node description contains the node name, node type, and bitwidth as follows:

```
node-name  node-type  bitwidth
```

Node-name is any 15 character name which is *unique* to the dataflow graph. The node-type is also a name of up to 15 characters which specifies the function of the node; this node-type MUST match one or more module functions. Bitwidth is a positive integer from 0 to whatever; a bitwidth of 0 informs **MAHA** that this node is an *implied* node (e.g. one that has no associated cost or delay). For example,

```
add1 add 8
```

names an adder *add1* which performs an *add* function and is of bitwidth *8*. There must be at least one *add* in the module library. Keep in mind that case is important; hence, *add* and *Add* are NOT the same.

A brief example of of some valid node names are shown below.

```
c3 dummy 1
x1_p1 bit-read 1
hello d-flop 140
me add 6
me_too sub 5
you-are-fired dummy 0
ADAM_rules or 2
Alice cmp 2
```

2.2.2 Edge Description

An edge description follows the node description in a dataflow file with an intervening blank line. It consists of a source node, destination node, and bitwidth.

```
source-node  destination-node  bitwidth
```

Like the node description, *source-node* and *destination-node* are names up to 15 characters in length. The node names MUST match names previously included in the node description.

An example of edge descriptions using previously listed node names are shown below.

```
hello me 8
hello Alice 16
Alice you-are-fired 16
ADAM_rules me_too 16
```

2.2.3 Module Description

The second input data file is the module library which consists of a list of individual functional modules and some of their physical parameters. The form of the module library is:

```
module-description-1
module-description-2
....................
module-description-p
```

Each *module-description* is of the form:

```
module-name module-function bit-width prop-delay cost
```

Module-name is a name of up to 15 characters which is *unique* to the module library. *Module-function* is also a name of up to 15 characters and describes the general function of the module. Some typical general functions are *add*, *sub*, *or*, *and*, and *d-flop*. The *node-type* of every node description MUST match the *module-function* of one or more modules. *Bit-width*, *prop-delay*, and *cost* are **integer** values which describe the bit width of the module, propagation delay in some arbitrary units, and cost (usually area, although could be power, etc.) in some arbitrary units.

When read into **MAHA**, the module library is reduced to a list which is linked to the node list. Specifically, for each *node-type* named in the node list, the module library is scanned as follows:

1. If the *node-type* and *module-function* are the same, proceed to step 2, else proceed to step 7.

2. If the *module bit-width* is less than or equal to zero, proceed to step 3, else proceed to step 5.

3. If the *module bit-width* is zero, the actual *cost* and *prop-delay* are defined as the *module cost* times the *node bit-width* and *module prop-delay* times the *node bit-width*, respectively. Proceed to step 6.

4. If the *module bit-width* is less than zero, the actual *cost* is the *module cost* times the *node bit-width* whereas the *prop-delay* is simply the *module prop-delay* (no adjustment). Proceed to step 6.
5. If the *module bit-width* is less than the *node bit-width*, proceed to step 7.
6. The *cost* and *prop-delay* information is added to the *matching module list*.
7. If there are more modules to check, proceed to step 1, else proceed to step 8.
8. An *average module* is created which has as its *prop-delay (cost)* the **average** of all prop-delays (costs) from the *matching module list*.

A pass over the module library is made for each node to create the *average module* which implements each node function. If two nodes generate the same matching module list, they will point to the same *average module*.

Since an *average module* is used by **MAHA**, a module library should contain identical functions with different prop-delays, costs, and bit-widths. If you wish to have direct control over the module library, then only a single compatible module should be included in the library for each node.

2.3 Running MAHA

This version of MAHA is written in C. To execute MAHA, type

```
maha.out
```

Once MAHA begins execution, it first inquires for the name of the dataflow description file.

```
Dataflow filename?
```

After you enter the name of your file, **MAHA** reads the node and edge list and attaches **root** and **outport** nodes if missing. Next, **MAHA** prompts for the module library.

```
Module library filename?
```

After the module library filename is entered, **MAHA** generates the *average module* library as described in the previous section. At this time, you are prompted for a number of self-explanatory items.

```
Echo to output file? (Y/N):

Print the node list? (Y/N):

Print the edge list? (Y/N):

Print the module library (Y/N):
```

For the first of the above inquiries, **MAHA** will prompt for a filename if you choose to echo the output to a file. Echoing is similiar to a script file as all console output is also directed to the specified file.

MAHA now calculates the **critical path** and lists the nodes in it, the total critical path time, and the minimum clock cycle time. (**Critical path** is the longest delay path from **root** to **outport**. Only a single critical path is returned even if there is more than one critical path.)

You are now asked for constraints which direct the search for a solution.

```
Enter maximum time (0 to search):

Enter maximum cost (0 to search):
```

There are three choices you have regarding the constraints assigned:

1. You can specify time (some positive integer) and set cost to 0 (to search). **MAHA** will search for the cheapest design which meets the time constraint.

2. You can specify cost (some positive integer) and time to 0 (to search). **MAHA** will search for the fastest design which meets the cost constraint.

3. You can specify BOTH cost and speed. **MAHA** will find the best design that meets both constraints. Due to the way **MAHA** functions, it will produce the fastest design that meets both of the constraints.

Notice that the case where BOTH maximum time and cost are set to zero (search) is NOT currently allowed. **MAHA** will print a message and ask for the time and cost again.

```
Do you wish to manually control the search? (Y/N):
```

The first big decision point has arrived. Normally, you will want the **MAHA** algorithm to find the result for you; in this case, answer **N** to the above question. If, for some reason, you wish to bypass the **MAHA** search algorithm, answer **Y** to the question. Since **MAHA** acts differently dependent on the answer, **manual** and **automatic** search are discussed separately.

2.3.1 MAHA Automatic Operation

When you specify automatic operation, **MAHA** will automatically serrch for the best result. However, **MAHA** can give a range of results

```
Do you just want the final result? (Y/N):
```

If you answer **N**, **MAHA** will give a table of results IGNORING the constraints. However, **MAHA** will stop when a solution is reached. Answering **Y** will not give the intermediate solution table, but executes faster since **MAHA** restarts allocation (at a higher partition count) when the cost (time) is exceeded rather than completing a "bad" allocation. The user can direct **MAHA** to reduce its execution time further.

```
Perform both ASAP (earliest) and ALAP (latest) allocation? (Y/N):
```

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP (default) allocation. When you first enter a graph or have a very large graph, it is recommended that you answer **N**. However, for small graphs, you may wish to try both despite the longer execution time. In this case, answer **Y**.

MAHA has the option to show all of its internal operation.

```
Show freedoms and status (detailed information)? (Y/N):
```

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: bounding the time range to search, buying and sharing of modules, current cost, and percent completion. If you wish to avoid this gory detail, enter **N** at the question.

MAHA pauses after allocating the to give the current cost. It then continues with off- allocation in the forward (ASAP) direction. **MAHA** also allocates in the reverse (ALAP) direction if instructed and takes the best of the two results.

2.3.2 MAHA Manual Operation

If manual search has been specified, the user directs all operations.

```
Enter partition count (positive #),
or clock cycle time (as negative #),
or <RETURN> to exit:
```

Manual control of **MAHA** allows either directly specifying the number of partitions (time slots) to cut the dataflow graph or entering the clock cycle time. (To distinguish a partition count from a clock cycle time, the former is entered as a negative number.)

A distinct feature of manual control of MAHA is that you are allowed to exceed the original constraints. MAHA will inquire whether to proceed when this happens.

2.3.2.1 Manual Partitioning

MAHA will partition a dataflow graph between 1 and *n* partitions, where *n* is the number of partitions realized when using the *minimum clock cycle time* discussed earlier. If the number of partitions is within this range, **MAHA** inquires

```
Perform both ASAP (earliest) and ALAP (latest) allocation? (Y/N):
```

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP (default) allocation. When you first enter a graph or have a very large graph, it is recommended that you answer **N**. However, for small graphs, you may wish to try both. In this case, answer **Y**.

MAHA has the option to show all of its internal operation.

```
Show freedoms and status (detailed information)? (Y/N):
```

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: **bounding the time** range to search, buying and sharing of modules, current cost, and **percent completion**. If you wish to avoid this gory detail, enter **N** at the question.

MAHA pauses after allocating the to give the current cost. It then continues with off- allocation in the forward (ASAP) direction. **MAHA** also allocates in the reverse (ALAP) direction if instructed and takes the best of the two results.

2.3.2.2 Manual Clock-cycle Entry

MAHA will partition a dataflow graph based on a preset clock cycle time. The only restriction is that the time must equal or exceed the *minimum clock cycle time* discussed earlier. If the clock cycle time is within this range, **MAHA** inquires

```
Perform both ASAP (earliest) and ALAP (latest) allocation? (Y/N):
```

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP (default) allocation. When you first enter a graph or have a very large graph, it is recommended that you answer **N**. However, for small graphs, you may wish to try both. In this case, answer **Y**.

MAHA has the option to show all of its internal operation.

```
Show freedoms and status (detailed information)? (Y/N):
```

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: bounding the time range to search, buying and sharing of modules, current cost, and percent completion. If you wish to avoid this gory detail, enter **N** at the question.

MAHA pauses after allocating the to give the current cost. It then continues with off- allocation in the forward (ASAP) direction. **MAHA** also allocates in the reverse (ALAP) direction if instructed and takes the best of the two results.

2.4 MAHA Output

Once **MAHA** has completed allocation of the graph, it displays the final clock cycle time, cost, and total time for the graph.

```
Show the hardware map? (Y/N):
```

If you wish to see the final allocated results, answer **Y** to the question. **MAHA** will output a table that looks like

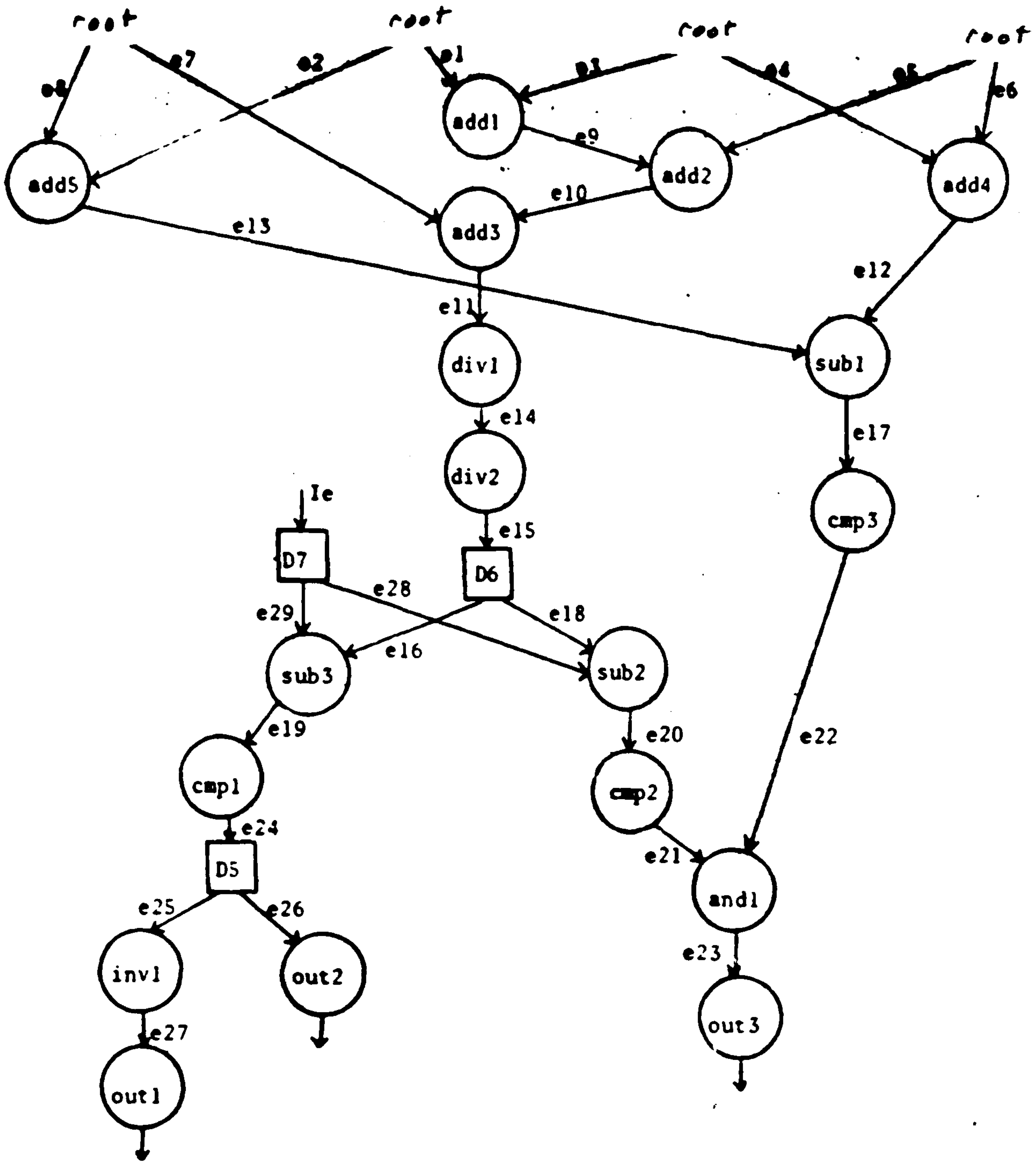
```
HARDWARE        NODE.SLOT
add8            add1.000        add5.001
r-shift10       div1.001
add8            add3.001
```

The first column contains the list of all hardware purchased, only the *function* is actually listed. (Notice there is one *r-shift10* and two *add8*s in the example.) At the columns to the right of the hardware is the list of all *nodes* which are bound to that piece of hardware and the time slot associated with it.

In the example, *add1* in the first partition (slot #0) and *add5* in the second partition (slot #1) share the same hardware - *add8*. *Add3* and *div1* were put into the second partition (slot #1) and do not share their hardware with any other operators.

2.5 An Example

In this section, the example that was included in the **MAHA** paper, "MAHA: A Datapath Synthesis Program" by Alice Parker, Jorge Pizarro, and Mitchell Mlinar, ACM/IEEE 23rd Design Automation Conference, June, 1986. The dataflow graph used in this example is reproduced below.



Dataflow Graph Example

Each node is assigned a distinct name (not to exceed 15 characters) for use by **MAHA**. Below is a copy of the dataflow graph file, *example.dfg*, which is accessible from the *maha* directory.

Since the paper was written, there have been some minor changes to **MAHA** - namely, the separation of conditional and parallel branches. The example dataflow graph in the paper has unconditional branches.

```
root dummy 0
outport dummy 0
add1 add 8
add2 add 9
add3 add 10
add4 add 9
add5 add 8
sub1 sub 9
sub2 sub 8
sub3 sub 8
div1 r-shift 10
div2 r-shift 10
cmp1 cmp 8
cmp2 cmp 8
cmp3 cmp 8
and1 and 2
inv1 inv 1
out1 buf 1
out2 buf 1
out3 buf 1
D5 parbeg 0
D6 parbeg 0
J5 parend 0
J6 parend 0

root add1 8
root add5 8
root add1 8
root add4 8
root add2 8
root add4 8
root add3 8
root add5 8
add1 add2 9
add2 add3 10
add3 div1 10
add4 sub1  9
add5 sub1  9
div1 div2  9
```

```
div2 D6 8
D6 sub3 8
sub1 cmp3 8
D6 sub2 8
sub3 cmp1 8
sub2 cmp2 8
cmp2 and1 1
cmp3 and1 1
and1 out3 1
cmp1 D5 1
D5 inv1 1
D5 out2 1
inv1 out1 1
out1 J5 1
out2 J5 1
out3 J6 1
J5 J6 1
J6 outport 1
```

Notice how the above example follows the rules outlined previously.

- there is a single **root** node which starts the dataflow graph
- there is a single **outport** node which ends the dataflow graph
- the *dummy*-type nodes have a bitwidth of 0. Since the **root** and **outport** nodes are algorithmic conveniences, a bitwidth of 0 informs **MAHA** to ignore and cost and delay for this node.
- there is a blank line between the node list and the edge list

The associated module library for this dataflow graph, *example.lib*, is reproduced below:

```
dummy dummy 0 0 0
parbeg parbeg 0 0 0
parend parend 0 0 0
add2 add 2 40 80
add4 add 4 72 120
add8 add 8 120 180
add12 add 12 150 220
add16 add 16 200 300
addn add 0 20 45
sub2 sub 2 50 90
sub4 sub 4 84 130
sub8 sub 8 140 200
sub12 sub 12 225 250
sub16 sub 16 240 360
subn sub 0 25 50
mul2 mul 2 80 140
mul4 mul 4 150 300
mul8 mul 8 280 640
mux2 mux 2 30 55
mux4 mux 4 54 100
cmp4 cmp 4 70 110
cmp8 cmp 8 130 180
cmp12 cmp 12 190 240
latch1 latch 1 30 68
latch4 latch 4 120 260
latch8 latch 8 240 500
latchn latch 0 30 66
and2 and 2 10 18
and3 and 3 14 22
r-shiftn r-shift 0 44 88
r-shift4 r-shift 4 44 250
r-shift8 r-shift 8 44 400
r-shift12 r-shift 12 44 510
r-shift16 r-shift 16 44 600
l-shiftn l-shift 0 44 88
l-shift8 l-shift 8 44 400
l-shift16 l-shift 16 44 600
inv1 inv 1 8 14
inv2 inv 1 8 25
buf1 buf 1 10 14
buf2 buf 1 30 100
buf3 buf 1 50 150
```

The sample module library points out some of the features and restrictions described earlier:

- Each module has a *unique* name.

- The **set of** module operations is well defined: *add*dition, *sub*tract, *mul*tiply, *cmp* (compare), *and*, *buf*fer driver, *inv*erter, *l-shift* (left shift register), *r-shift* (right shift register/divider), *dist*ribute, and *join*.

- Even fictitious node operations such as *parbeg*, *dummy*, and *parend* MUST be declared in the module library. (Other fictitious nodes include *dist* and *join*, but are not used in this example.)

- All of the delay and cost values are positive integers (including zero).

Here is a sample run of **MAHA** using the example.

```
sun[1] maha.out



                    MAHA v5.01
              USC Design Automation Group

             Mitchell Mlinar, October 1986


Dataflow graph filename? dataflow

Reading in the nodelist.
There are 24 nodes: roots = 1, outports = 1.

Reading in edgelist.
There are 32 edges.
Checking for extra edges required.
—> 0 extra edges added.

Module library filename? modlib

There are 13 modules, minimum possible time is 230.
Input process time:  0.180 seconds.

Echo to output file? (Y/N): n

Print the node list? (Y/N): n

Print the edge list? (Y/N): n

Print the module library? (Y/N): y
```

```
Module name     Width     Delay     Cost
dummy0              0         0        0
add8                8       157      265
add9                9       176      308
add10              10       183      323
sub9                9       230      353
sub8                8       201      302
r-shift10          10       176      663
cmp8                8       160      210
and2                2        12       20
inv1                1         8       19
buf1                1        30       88
parbeg0             0         0        0
parend0             0         0        0


Press RETURN to continue --
```

Notice how **MAHA** calculates the *average* of the module library. Each node in the dataflow will have a single module associated with it (but not alloacted yet).

```
Finding the critical path.

The critical path has 13 nodes with a time of 1271.
The minimum clock time is 230.
Critical path process time:  0.020 seconds.


The critical path is:

root          add1          add2          add3
div1          div2          D6            sub2
cmp2          and1          out3          J6
outport



Enter maximum time (0 to search):
```

Now that the critical path has been found, we can try to perform the synthesis with a cost constraint.

```
Enter maximum time (0 to search): 0

Enter maximum cost (0 to search): 3000



Constraints:
   Time: minimize   Cost: 3000

Do you wish to manually control the search? (Y/N): n


Automatic search ...


Perform both ASAP (earliest) and ALAP (latest) allocation? (Y/N): y

Show freedoms and status (detailed information)? (Y/N): n


Partitions  Clock     Time    Cost
     1       1271     1271    4685
     2        692     1384    3449
     3        516     1548    4475
     4        377     1508    3105
     5        352     1760    4475
     6        333     1998    3812
     7        230     1610    3504


Best is time of 1508, clock of 377, cost of 3105

Analysis time:  0.280 seconds.

Recalculate the best case showing the hardware map? (Y/N)  y
```

```
HARDWARE        NODE.SLOT
add8            add1.000
add9            add2.000
add10           add3.001
r-shift10       div1.001           div2.002
sub8            sub2.002
cmp8            cmp3.002           cmp2.003
and2            and1.003
buf1            out3.003
sub9            sub1.001
add9            add4.000
add8            add5.000


Bye.
sun[2]
```

Since the cost constraint was never met, the solution with the lowest cost was selected. Obviously, the graph is too tightly constrained, so a higher cost will be attempted. (Of course, in this simple example, the solution is obvious. In a large example, the solution may not be easily seen.)

```
Enter maximum time (0 to search): 0

Enter maximum cost (0 to search): 4000


Constraints:
   Time: minimize   Cost: 4000

Do you wish to manually control the search? (Y/N): n


Automatic search ...


Perform both ASAP (earliest) and ALAP (latest) allocation? (Y/N): n

Show freedoms and status (detailed information)? (Y/N): n


Partitions  Clock    Time    Cost
    1       1271     1271    4685
    2        692     1384    3449
    3        516     1548    4475
    4        377     1508    3105
    5        352     1760    4475
    6        333     1998    3812
    7        230     1610    3504


Best is time of 1384, clock of 692, cost of 3449

Analysis time:  0.080 seconds.

Recalculate the best case showing the hardware map? (Y/N): y
```

```
HARDWARE      NODE.SLOT
add8          add1.000       add5.001
add9          add2.000       add4.001
add10         add3.000
r-shift10     div1.000       div2.001
sub8          sub2.001
cmp8          cmp2.001
and2          and1.001
buf1          out3.001
sub8          sub3.001
cmp8          cmp1.001
inv1          inv1.001
buf1          out1.001
buf1          out2.001
sub9          sub1.001
cmp8          cmp3.001


Bye.
sun[2]
```

Although the LOWEST cost is for 4 partitions, the object was to *minimize* time while meeting the cost constraint. Note that nodes with no cost associated with them are not listed in the hard ware map.

3. Converting to PNF from a VT description

Converting a complete **VT** (Value-Trace) description for a datapath into **PNF** (Park Normal Form) needed by **MAHA** is a four step process requiring the following programs:

vt-pre.out	The VT pre-processor
vtran.l	The VT translator
makpnf.l	The PNF extracter
cnvrtm.l	The LISP-to-C data translator

3.1 VT Pre-processor: vt-pre.out

The VT pre-processor merely reads in the original VT dsecription and writes out an easily LISP readable VT description. **Vt-pre.out** removes all of the unnecessary declarations and inserts field delimiters if they do not currently exist (as astericks). To execute the pre-processor, enter:

```
vt-pre.out input-VT-description processed-VT-description
```

where *input-VT-description* and *processed-VT-description* are the input and output filenames, respectively. For example, assume that you wish to synthesize a datapath for *vt181.ibm*. You would type:

```
vt-pre.out vt181.ibm vt181.pre
```

The pre-processed VT description would be written to *vt181.pre*.

3.2 VT Translator: vtran.l

The VT translator consists of a single LISP module called **vtran.l**. This can be loaded by typing:

```
lisp vtran
```

Once the load is complete, the translator can be executed by typing:

```
(G)
```

Upon execution, the VT translator prints a sign-on message and asks for the name of the input file. This would normally be the output from the VT pre-processor. **vt-pre.out:**

```
Input file? vt181.pre
```

Next, the VT translator inquires for any starting list. For extremely complex VT descriptions, it is wise to break the VT up into several smaller files. Since the translation process is not quick, you could incrementally translate the pieces and quit after each one saving the intermediate results. (You can always do a large VT as a single module. However, since humans have explicit needs such as food, water, and sleep and generally like to be aware of a problem in a program today rather than sometime next week, you are forewarned.) For this example, there is no starting list:

```
Any starting list? n
```

At this point, the VTs are processed one at a time showing the translation as it proceeds. The sequential translation has exception handling for four special cases: *CALL*, *ENTER*, *RESTART*, and *LEAVE*. When a *CALL* occurs, the current *translated* VT list is examined for the presence of the named VT operator. If the VT body has been translated, the VT body is substituted for the call (flattening of the hierarchy) and the translation continues. However, if the VT body has *not* been encountered, the translation of the current VT is aborted and the program proceeds to the next VT in the input file. Although this method means that a multi-pass approach may be necessary to generate a complete translation, there is much less overhead involved (which is often critical in LISP).

The *ENTER* instruction is treated identical to a *CALL* instruction in this version of **vtran.l**.

The *LEAVE* instruction is only accepted at the end of a VT in this version of **vtran.l**. Hence, any internal *LEAVEs*, regardless of whether they leave the current VT or another VT, must be removed. If a VT has outputs any values required by other VTs, a *LEAVE* with the appropriate values must be the last VT-body statement.

Once the first pass through the VT is complete, the program checks if all VT bodies have been translated. If not, another pass is made through the file; however, all VT bodies which have been previously translated are skipped. The translator will continue to make passes through the file until no more VTs can be resolved or the translation is complete. If the translation is complete, completion status is displayed.

```
translation process complete: xxx VT bodies translated
```

If the translation process could not complete, a message like:

```
translation process suspended: xxx VT bodies translated
                               yyy unresolved VT bodies
```

After printing the pertinent message, the translator asks if you would like to process any other files. Note that the results of the new file are incrementally added to the current description: if you want to translate a completely unrelated VT, you must stop the program and start over.

```
Any more files? n
```

Since the translation is being stopped (it does not matter whether the translation is complete), the program inquires:

```
Output filename? vt181.trn
```

The translator will write the translated portions of the VT description out to the named file and exit.

If the translation is complete, you can proceed to generate the PNF data lists (via **makpnf.l** discussed next). Otherwise, **vtran.l** will have to be executed using *vt181.trn* as the starting list and one or more files containing the remainder of the VTs as the input file(s).

3.3 Extracting the PNF datafiles: makpnf.l

Once the VT translator has successfully terminated, the desired VT can be extracted from the translation file. To load the extracter, type:

```
lisp makpnf
```

Once the load is complete, you can execute the program by typing:

```
(G)
```

After printing the signon message, the extracter will inquire for the file which was generated by the translator:

```
File generated by VIRAN? vt181.trn
```

After reading the translated VT descriptions, the VT body to be extracted is entered:

```
VT body to convert to PNF? ?
```

Since remembering (or even knowing) the names of all VT bodies is, at best, ridiculous, entering a question mark instructs the extracter to list the names of VTs in this file.

```
    List of VT bodies available:
v181  v184  v186  v113
```

You have chosen to synthesize what your filename indicates is the "big cheese", so you enter:

```
VT body to convert to PNF? v181
```

After selecting the VT body to extract, enter the nodelist and edgelist filenames you wish to generate:

```
Nodelist file to create? vt181.nod

Edgelist file to create? vt181.edg
```

The files output by **makpnf.l**, *vt181.nod* and *vt181.edg*, are written in LISP format as early versions of **MAHA** were composed in LISP. These files need to be converted to the C text format before executing the C version of **MAHA**.

3.4 LISP-to-C MAHA converter: cnvrtm.l

Cnvrtm.l is a LISP program which converts from the early nodelist and edgelist descriptions in LISP format to a single dataflow graph in C. The reverse conversion from C to LISP data files is not supported. To load the conversion program, type

```
lisp cnvrtm
```

Once loading has completed, the program is executed with

```
(G)
```

Upon execution, **cnvrtm.l** prints a signon message and inquires for the names of the LISP format nodelist and edgelist files (generated by **makpnf.l**).

```
LISP format nodelist filename? vt181.nod

LISP format edgelist filename? vt181.edg
```

Once the lists are read, **cnvrtm.l** inquires for the output filename.

```
Dataflow output filename? vt181.dfg
```

The dataflow graph *vt181.dfg* is written and **cnvrtm.l** exits. After construction of a compatible module library (say, *vt181.lib*), you have the two input data files necessary for executing **MAHA**.

4. File Formats

Included here is a brief summary of all file formats, some of which are used by the C version of **MAHA** and others which are used by LISP utilities.

4.1 Dataflow description file: C

The *dataflow description file* has both node and edge information in a one node (edge) per line format as follows:

```
node-description-1
node-description-2
..................
node-description-n

edge-description-1
edge-description-2
..................
edge-description-m
```

Note the blank line between the node and edge descriptions; this is a REQUIREMENT.

4.1.1 Node Description

A node description contains the node name, node type, and bitwidth as follows:

```
node-name  node-type  bitwidth
```

Node-name is any 15 character name which is *unique* to the dataflow graph. The node-type is also a name of up to 15 characters which specifies the function of the node; this node-type MUST match one or more module functions. Bitwidth is a positive integer from 0 to whatever; a bitwidth of 0 informs **MAHA** that this node is an *implied* node (e.g. one that has no associated cost or delay). For example,

```
add1 add 8
```

names an adder *add1* which performs an *add* function and is of bitwidth *8*. There must be at least one *add* in the module library. Keep in mind that case is important; hence, *add* and *Add* are NOT the same.

A brief example of of some valid node names are shown below.

```
c3 dummy 1
x1_p1 bit-read 1
hello d-flop 140
me add 6
me_too sub 5
you-are-fired dummy 0
ADAM_rules or 2
Alice cmp 2
```

4.1.2 Edge Description

An edge description follows the node description in a dataflow file with an intervening blank line. It consists of a source node, destination node, and bitwidth.

```
source-node  destination-node  bitwidth
```

Like the node description, *source-node* and *destination-node* are names up to 15 characters in length. The node names MUST match names previously included in the node description.

An example of edge descriptions using previously listed node names are shown below.

```
hello me 8
hello Alice 16
Alice you-are-fired 16
ADAM_rules me_too 16
```

4.2 Dataflow description file: LISP

The LISP dataflow graph is similiar to the C version with some additions necessary for LISP and interchangability with SEHWA and CSSP. The dataflow graph file consists of a LISP node list followed by a LISP edge list.

```
((node-description-1)
 (node-description-2)
 (..................)
 (node-description-n))
((edge-description-1)
 (edge-description-2)
 (..................)
 (edge-description-m))
```

Each type will be described next.

4.2.1 Node list description

The node description is a LISP list of the form:

```
(node-description-1 node-description-2 ..... node-description-n)
```

where *node-description-i* is a list of the form:

```
(node-name node-function bit-width)
```

where *node-name* is any node-list unique character string from 1 to 31 characters and the first character is a letter (a-z). *node-function* is the function of the node (character string from 1 to 31 characters) and should be consistent for all nodes. Some common functions are: *add*, *sub*, *div*, *mul*, *d-flop*, and *dummy*. The only restriction on the function name is that it MUST match at least one function in the module library. *Bit-width* is an integer from 0 to whatever and defines the minimum bit width needed for this function.

The only characters which may comprise a character string in the dataflow description are the letters (A-Z, a-z, 0-9, - and _). Keep in mind that case is important; hence, *add* and *Add* are NOT the same. Also, long and fancy names such as *lets-go_get_a_six-pack* may be cute, but they do take up precious memory in LISP.

A brief example of a node list is shown below.

```
((c3 dummy 1)
 (x1_p1 bit-read 1)
 (hello d-flop 140)
 (me add 6)
 (me_too sub 5)
 (you-are-fired dummy 0)
 (ADAM_rules or 2)
 (so_does_Alice cmp 2))
```

4.2.2 Edge list description

An edge description is a LISP list of the form:

```
(edge-description-1 edge-description-2 ..... edge-description-n)
```

where *edge-description-i* is a list of the form:

```
(edge-name source-node destination-node bit-width edge-name)
```

where *edge-name* is any edge-list **unique** character string from 1 to 31 characters and the first character is a letter (a-z). *Source-node* and *destination-node* are the source and destination nodes of the directed arc. Both node names MUST match names used in the node list description. Not only is it unwise, unfair, and immoral to use a name not in the nodelist, the utilities using the file will not work if there is no match. *Bit-width* is an integer from 1 to whatever and defines the minimum bit width needed for this arc.

(Edge-name is repeated at the end for compatibility with some other portions of the ADAM system.)

A brief example of an edge list is shown below.

```
((e1 v24_11 x1_p1 1 e1)
 (e2 c3 x1_p1 1 e2)
 (ez x1_p1 x2_p1 1 ez)
 (zzzzzz v24_12 x3_p1 6 zzzzzz)
 (wake-up bozo time 2 wake-up)
 (snort_yawn_huh what_did you_wake_me 4 snort_yawn)
 (arghhhhhh x51_p1 pf 1 arghhhhhh))
```

4.3 Module description file: C

The module library data file consists of a list of individual functional modules and some of their physical parameters. The form of the module library is:

```
module-description-1
module-description-2
....................
module-description-p
```

Each *module-description* is of the form:

```
module-name module-function bit-width prop-delay cost
```

Module-name is a name of up to 15 characters which is *unique* to the module library. *Module-function* is also a name of up to 15 characters and describes the general function of the module. Some typical general functions are *add*, *sub*, *or*, *and*, and *flip-flop*. The *node-type* of every node description MUST match the *module-function* of one or more modules. *Bit-width*, *prop-delay*, and *cost* are **integer** values which describe the bit width of the module, propagation delay in some arbitrary units, and cost (usually area, although could be power, etc.) in some arbitrary units.

[illegible] read into **MAHA**, the module library is reduced to a list which is linked to the [illegible].

4.4 Module description file: LISP

LISP data structure for the module library is nearly identical to the C version with the exception of the addition of parentheses.

```
(module-description-1 module-description-2 ... module-description-n)
```

Each *module-description-i* is of the form:

```
(module-name module-function bit-width prop-delay cost)
```

See module library description for a description of each of the subparts.

IV. PLEST: An Area Estimator for Polycell Chips

```
*******************************************************************************
*
*              PLEST : An area estimator [illegible]
*                      Author : [illegible]
*
*      INPUTS :    total cell width [illegible]
*                  number of equivalent [illegible]
*                  average wire length [illegible]
*                  max and min number of rows [illegible]
*
*      OUTPUTS :   table of estimates [illegible]
*                          # of rows
*                          [illegible]
*                          chip height
*                          chip width
*                          chip aspect ratio
*                          chip area
*                    if requested the program [illegible]
*                    for any two of the above [illegible] parameters
*
*      NOTES :     the program [illegible]
*                  for other systems the [illegible]
*                  should be changed [illegible]
*                  the width of the chip [illegible]
*                  width row [illegible] estimated max [illegible]
*                   (fw*maxfeed) [illegible]
*                  the height of the chip [illegible]
*                  height [illegible]
*
*      To run PLEST :
*                  type [illegible]
*******************************************************************************
```

END

DATE
FILMED

5-87

[illegible]